A Modeling Approach based on UML/MARTE for GPU Architecture

Antonio Wendell de Oliveira Rodrigues, Frédéric Guyomarc'H, Jean-Luc Dekeyser

▶ **To cite this version:**

Antonio Wendell de Oliveira Rodrigues, Frédéric Guyomarc'H, Jean-Luc Dekeyser. A Modeling Approach based on UML/MARTE for GPU Architecture. Symposium en Architectures nouvelles de machines (SympA'14), May 2011, Saint Malo, France. inria-00593863

HAL Id: inria-00593863
https://inria.hal.science/inria-00593863

Submitted on 19 May 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

RenPar'20 / **SympA'14** / CFSE 8
Saint-Malo, France, du 10 au 13 mai 2011

A Modeling Approach based on UML/MARTE for GPU Architecture

Antonio Wendell de O. Rodrigues, Frédéric Guyomarc'h and Jean-Luc Dekeyser

LIFL - USTL
INRIA Lille Nord Europe - 59650
Villeneuve d'Ascq - France
wendell.rodrigues@inria.fr

Abstract
Nowadays, the High Performance Computing is part of the context of embedded systems. Graphics Processing Units (GPUs) are more and more used in acceleration of the most part of algorithms and applications. Over the past years, not many efforts have been done to describe abstractions of applications in relation to their target architectures. Thus, when developers need to associate applications and GPUs, for example, they find difficulty and prefer using API for these architectures. This paper presents a metamodel extension for MARTE profile and a model for GPU architectures. The main goal is to specify the task and data allocation in the memory hierarchy of these architectures. The results show that this approach will help to generate code for GPUs based on model transformations using Model Driven Engineering (MDE).

Keywords : GPU, Embedded Systems, MDE, Code Generation

1. Introduction

Over the past years, software researchers and developers have been creating abstractions that help them program in terms of their design intent rather than the underlying architectures, e.g., CPU, memory, and network devices. Moreover, they shield themselves from the complexities of these architectures.

Today, developers need to specify their software based on used platform architectures. Model Driven Architecture (MDA)[5] is a framework proposed by Object Management Group (OMG) for software development. This framework is driven by models in different abstraction levels. Approaches based on MDA have been used as a solution to accelerate the embedded system design. In this solution, initially a system is modeled using a Platform-Independent Model (PIM). Then, using transformation languages, this model is transformed in a Platform-Specific Model (PSM).

In order to define these models, the OMG proposes Meta Object Facility (MOF)[3] and Unified Modeling Language (UML) as standard for modeling and meta-modeling. Furthermore, the OMG adds the profile concept which provides a generic extension mechanism for customizing UML models for particular domains and platforms. Modeling and Analysis of Real-time and Embedded systems (MARTE)[1] is a profile that adds capabilities to UML for model-driven development of Real Time and Embedded Systems (RTES).

Lately, a number of researches in High Performance Computing (HPC) have led to use of Graphics Processing Unit (GPU). GPU is a manycore processor attached to a graphics card dedicated to calculating floating point operations. Initially, the GPU was dedicated to graphics operations in personal and desktop computers. But, its processing power allowed executing other algorithms. This was called General Purpose GPU (GPGPU). These processors have been progressively part of embedded systems [10]. Thus, in order to optimize their application performance, embedded systems can use the processing power of GPUs. In this paper, we present a metamodel for designing of GPU characteristics, and a model for a specific GPU architecture. Both metamodel and model are part of *Gaspard2*[8], which is an environment for development of RTES and Intensive Signal Processing (ISP) applications. *Gaspard2* is the flagship project of DaRT team.

We divide this paper in eight sections. The section 2 shows the main aspects of model driven engineering. The next one presents the purpose of MARTE profile. The GPU architecture is described in the

section 4. Then, in the next section we show our metamodel approach. The section 6 depicts an example model for devices based on GPU. Finally, in the section 7 and 8, we presents the environment of development with a case study and conclusions respectively. in

2. Model Driven Engineering

Model Driven Engineering (MDE) is a software development methodology which focuses on creating models, or abstractions, closer to some particular domain concepts rather than computing (or algorithmic) concepts. It is meant to increase productivity by maximizing compatibility between systems, simplifying the process of design, and promoting communication between individuals and teams working on the system. The best known MDE initiative is the MDA, which is a registered trademark of OMG.

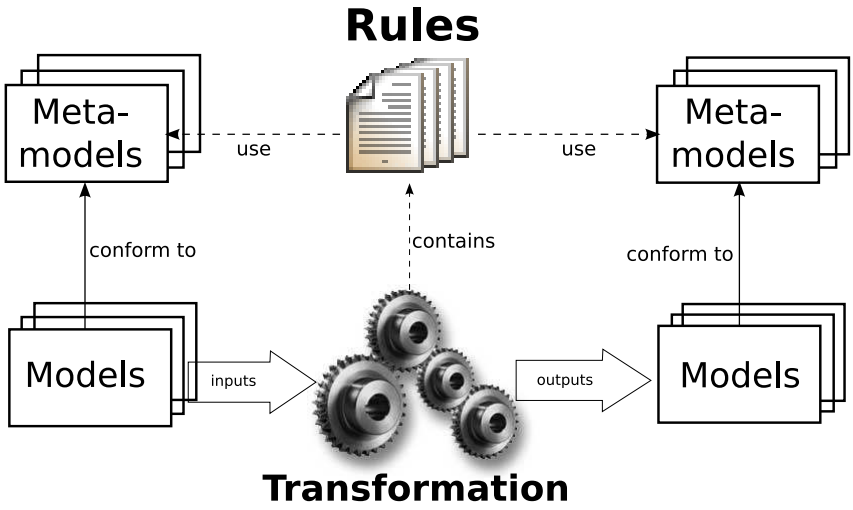

FIGURE 1 – Model Transformation Illustration

In MDE, complex systems can be easily understood thanks to abstract and simplified representations : models. Graphical representations of models considerably facilitate the comprehension of a given model. The UML language has been often used for such graphical representations since its normalization in 1997. A model highlights the intention of a system without describing the implementation details. MDE is definitely oriented towards the modeling of software engineering systems. The resulting models must be comprehensive and interpretable by computer. In addition, in MDE, we can use Domain-Specific Modeling Language (DSL) to model systems with peculiar characteristics. MDE also covers the code generation, which puts a model in concrete form. In this way, MDE stands apart from the others methodologies based on models. The next subsections detail the major aspects of MDE that are model, metamodel and model transformations.

2.1. Model

A model is an abstraction of the reality. Models are composed of concepts and relations. The concepts represent an abstraction of objects and relations represent the links between the objects. In addition, models can be graphically observed from different points of view (views in MDE), which highlight specific aspects of the reality.

Applications and architectures of embedded systems have clearly identified elements (objects) such as data parallel tasks, data dependencies, multidimensional data arrays, and architecture parts. The abstraction of each element corresponds to a concept in a model, the dependencies between these elements are represented by relations. Models can represent abstract descriptions of these applications and thus, it helps to specify and modify them since each concept and relation are clearly identified. Moreover,

views can help to represent and document models by highlighting the relevant concepts and relations according to a particular purpose.

2.2. Metamodel

A metamodel gathers the set of concepts and relations between the concepts used to describe a model, i.e., the reality according to a particular purpose (a given abstraction level for instance). Then a model conforms to a metamodel which specifies a modeling structure. In the other words, a metamodel defines the syntax of its models, like a grammar defines its language. Consequently, a metamodel can state the set of necessary concepts and relations to represent the applications and architectures of embedded systems at a given abstraction level. A model always conforms to a metamodel. This relation is called conformance. The conformance relation has a different nature than the representation relation between a model and its system. A metamodel does not represent a model (that could be considered a system), but only the concepts and relationships that may be created. Additionally, a metametamodel defines the syntax of its metamodel, then a metamodel conforms to a metametamodel.

2.3. Model Transformations

In MDE, a model transformation is a compilation process which transforms a source model into a target model. The source and the target models are respectively conformed to the source and the target metamodels (fig. 1). A model transformation relies on a set of rules. Each rule clearly identifies concepts in the source and the target metamodels. Such decomposition facilitates the extension and the maintainability of a compilation process : new rules extend the compilation process and each rule can be modified independently from the others. The rules are specified with languages. The language may be imperative : it describes how a rule is executed ; it can be declarative, it describes what is created by the rules. Declarative languages are often used in MDE because the rules objectives can be specified independently from the execution. A graphical representation is a good approach for representing the rules expressed in a declarative language.

3. MARTE Profile

The UML profile for MARTE (or MARTE profile) extends the possibilities for modeling of application and architecture and their relations. In addition, MARTE allows extending the performance analysis and task scheduling based on target platform architecture.

MARTE consists in defining foundations for model-based description of real time and embedded systems. These core concepts are then refined for both modeling and analyzing concerns. Modeling parts provide support required from specification to detailed design of real-time and embedded characteristics of systems. MARTE concerns also model-based analysis. In this sense, the intent is not to define new techniques for analyzing real-time and embedded systems, but to support them. Hence, it provides facilities to annotate models with information required to perform specific analysis. Especially, MARTE focuses on performance and schedulability analysis. But, it defines also a general analysis framework which intends to refine/specialize any other kind of analysis. Among others, the benefits of using this profile are thus :

- providing a common way of modeling both hardware and software aspects of a RTES in order to improve communication between developers ;
- enabling interoperability between development tools used for specification, design, verification, code generation, etc. ;
- fostering the construction of models that may be used to make quantitative predictions regarding real-time and embedded features of systems taking into account both hardware and software characteristics.

The fig. 2 shows the MARTE profile architecture. Moreover, we highlight some of our contributions within DaRT team. Over the last years, the DaRT team has contributed to specify the *Allocation Modeling* (Alloc), *Generic Component Modeling* (GCM), *Hardware Resource Modeling* (HRM), *Value Specification Language* (VSL), and *Repetitive Structure Modeling* (RSM). In this paper, we focus on the Alloc, HRM and RSM components. In addition, we purpose a *memory mapping* metamodel in order to describe the data allocations in the memory.

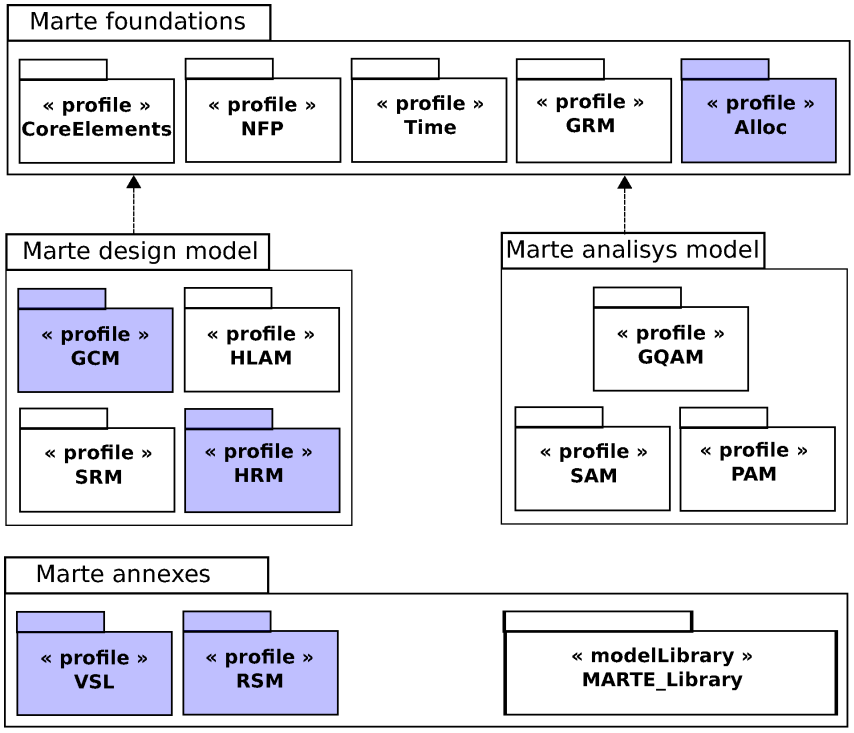

FIGURE 2 – Architecture of the MARTE profile

4. Graphics Processing Unit

The GPU devotes more transistors to data processing rather than data caching and flow control. This is the reason why the GPU is specialized for compute intensive. NVIDIA GPUs, more precisely, are composed of array of Streaming Multiprocessors (SM or Compute Units), which is equipped with 64 scalar cores (the SP, Streaming Processors or Processing Elements), 16834 32-bit registers, and 48KB of high-bandwidth low-latency memory shared for up to 1024 co-resident threads (or work-items). GPUs such as the NVIDIA GeForce GTX 480 contain 15 Streaming Multiprocessors, each of which supports up to 1024 co-resident threads, so 30K threads can be created for a certain task. In addition, each multiprocessor executes groups, called warps, of 32 threads simultaneously. NVIDIA's actual CUDA architecture, code-named Fermi, has features for general-purpose computing. Fundamentally, Fermi processors are still graphics processors, not general-purpose processors. The system still needs a host CPU to run the operating system, supervise the GPU, provide access to main memory, present a user interface, and perform everyday tasks that have little or no data-level parallelism.

4.1. Memory Architecture

In the NVIDIA GPU memory hierarchy, there are per-thread local, per-block shared, and device memory which comprehend global, constant, and texture memories. Shared memory can be only accessed by threads in the same block. The shared memory space is much faster than the local and global memory spaces due to placement on chip. But, except for Fermi, only 16KB of shared memory are available on each SM.

4.2. Programming Model

Compute Unified Device Architecture (CUDA) is a C language extension developed by NVIDIA to facilitate writing programs on GPUs. This extension allows the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. One of the main features of CUDA is the provision of a Linear Algebra library (CuBLAS) and a Fast Fourier Transform library(CuFFT) [2]. Khronos Group

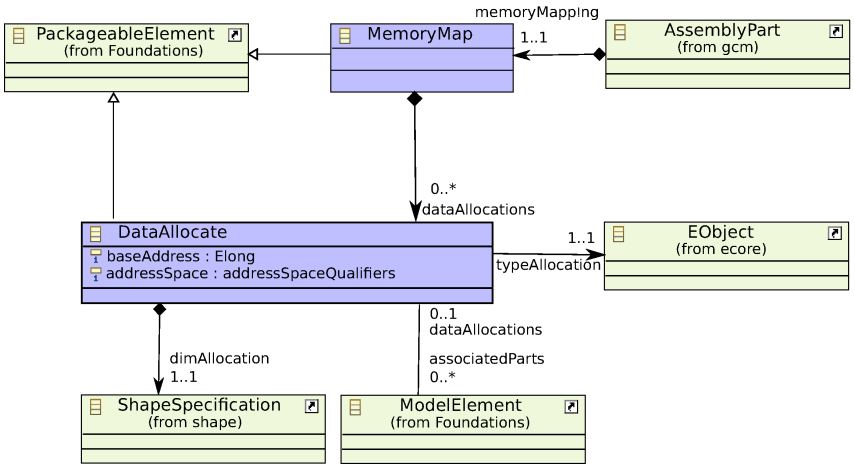

FIGURE 3 – Memory Mapping Metamodel Extension for MARTE

has released the specification to OpenCL (currently 1.1). OpenCL (Open Computing Language) is the an open, royalty-free standard for general-purpose parallel programming of heterogeneous systems. It provides an uniform programming environment for software developers to write efficient, portable code for high-performance computing servers, desktop computer systems and handheld devices using a diverse mix of multi-core CPUs, GPUs, Cell-type architectures and other parallel processors such as DSPs. The OpenCL has some similarities with CUDA programming model [4].

5. MARTE Metamodel Extensions

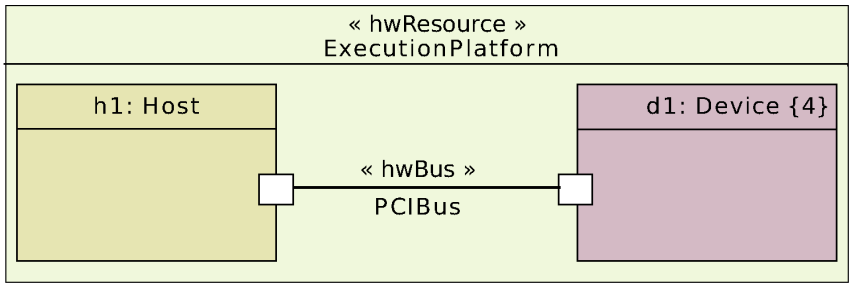

FIGURE 4 – Host and Device Architecture

The UML Tools provide a standard representation of UML diagrams and concepts. UML also allows to extend these concepts by the use of profiles and stereotypes. Actually, representation of these stereotypes can be customized in a limited way. MARTE profile provides us stereotypes to describe new elements in an UML model. Stereotypes are one of three types of extensibility mechanisms in UML. They allow designers to extend the vocabulary of UML in order to create new model elements, derived from existing ones, but that have specific properties that are suitable for a particular problem domain or otherwise specialized usage. There are two ways for defining UML-based Domain-Specific Modeling Languages (DSL). In the first one, DSL can be defined as a UML profile, such as MARTE profile, in a lighweight way, using stereotypes and tagged values. In the second way, the Meta Object Facility (MOF) can be used to either extend the UML metamodel, or to direcly define the metamodel without dependency on UML. We chose the second one for extending MARTE. The purposed metamodel defines an abstract syntax for our DSL.

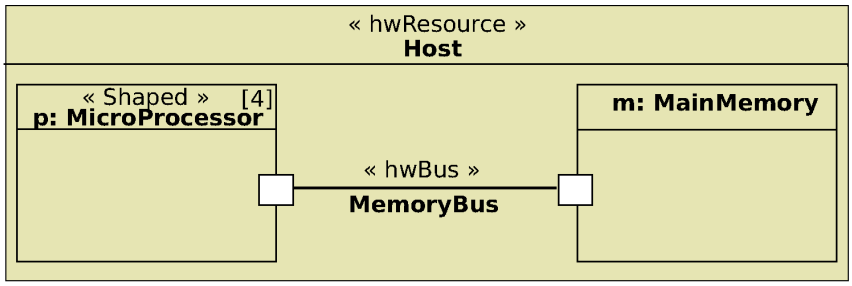

FIGURE 5 – Host Components

The DaRT team has worked on metamodel extensions for MARTE. Among these extensions, we specialized an extension for memory concepts in modeling GPU architecture. The fig. 3 shows the abstract syntax that defines memory mapping concepts. Two new classes were created to add concepts for data allocation over a memory map : *MemoryMap* and *DataAllocate*. An *AssemblyPart*, generally an instance of memory component in the defined model, should have one *memoryMapping* of *MemoryMap* type which is composed or not by *dataAllocations* of the *DataAllocate* type. Each *dataAllocation* has its data scopes. This scope (***spaceAddress***) is defined by ***addressSpaceQualifiers = {global,constant,local,private}*** such as speficied in GPU memory hierarchy. Additionally, the *dataAllocation* has :

1. ***baseAddress*** : of the long integer data type and it specifies the base reference value or the base pointer of the variable ;
2. ***dimAllocation*** : this information comes from *flowPorts shape* in the model and it defines the allocation size ;
3. ***associatedParts*** : it lists all the elements in the model that use the same allocation ;
4. ***typeAllocation*** : this information comes from *flowPorts type* in the model and it defines the variable datatype.

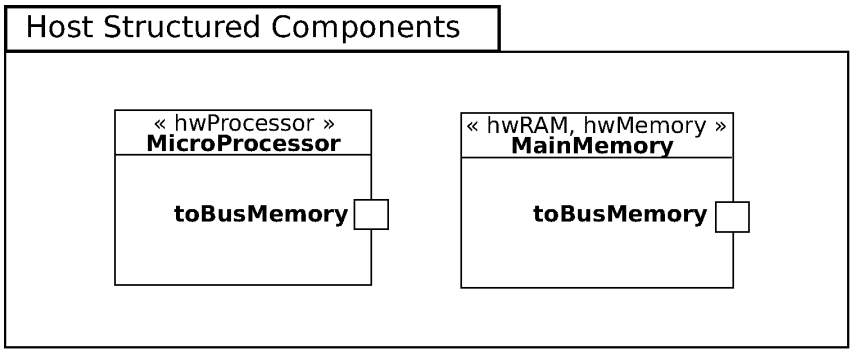

FIGURE 6 – Host Structured Components Package

This metamodel, also known as *MemoryMapping* metamodel extension for MARTE, allows us to specify variables and their attributes. We can obtain the information from *flowPort* elements which are specified in the application model.

6. GPU Architecture Models

6.1. Platform Model

In order to run applications on GPUs, developers should take into account that GPUs do not work alone. GPUs are coprocessors that need a host. Nowadays, CPUs with applications based on C language are

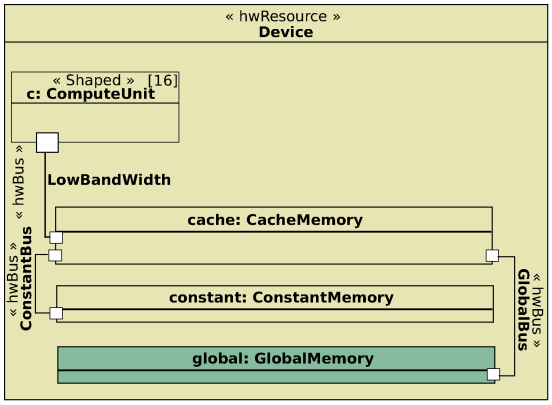

FIGURE 7 – Device Internal Model

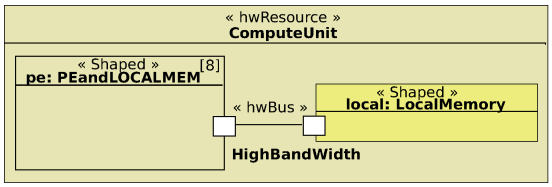

FIGURE 8 – Internal Structure of Compute Units

the most suitable host for GPUs. The fig. 4 depicts the host and device modeling. A databus, generally a *pcibus*, links host and device. The host uses this bus to communicate with the device for threads(work-items) launches and data copies. In our model, the host is divided in two component instances of *MicroProcessor* and *MainMemory*. The first one is a component that represents the processor and it has a *hwProcessor* stereotype (fig. 6) from MARTE profile. This stereotype allows specifying processor details, such as `speedFactor`, `frequency` or `isActivate` status. In addition, the *MicroProcessor* instance can be stereotyped with *Shaped*. With this stereotype, the instance can provide multiplicity informations. The fig. 5 shows a shaped processor with a single dimension multiplicity which is equal to 4. The *Shaped* stereotype has an important role in application task distribution over processors. The *hwProcessor* and *Shaped* stereotypes are described in HRM and RSM packages of MARTE.

The device in the architecture model is the GPU. As our intent is modeling GPU architecture, we add more information about the interation between components of its structure. Since CUDA is proprietary, we opted for OpenCL standard [4]. In OpenCL, a device is divided into one or more compute units (CUs) which are further divided into one or more processing elements (PEs). Computations on a device occur within the processing elements. In the presented model in fig. 7, we have an instance *c* of the *ComputeUnit* type and *Shaped* value is equal to 16. Each *ComputeUnit* is composed of 8 *ProcessingElement* as described in fig. 8 and fig. 9.

6.2. Memory Model

In order to obtain a clear and useful model, we specified a simple memory structure in the host side. The host has one single shared memory component. This component is *hwMemory* or *hwRAM* stereotyped, thus we can map *flowPorts* over this memory instance and obtain allocations for application variables.

However, the device has more complex memory structure. There is a hierarchical memory composition that allows different visibility and bandwidth.

1. ***Global Memory*** : work-items(threads on NVIDIA definition) can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device ;

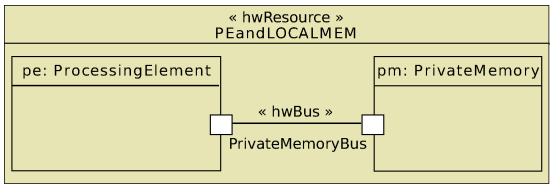

FIGURE 9 – Internal Structure of Processing Elements

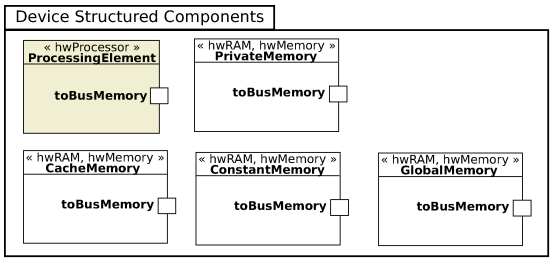

FIGURE 10 – Device Components

2. ***Constant Memory*** : a region that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory ;
3. ***Local Memory*** : a memory region local to a compute unit. This memory region can be used to allocate variables that are shared by all work-items placed on the same compute unit. It may be implemented as dedicated regions of memory on the device ;
4. ***Private Memory*** : a region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item. Each processing element has its private memory as illustrated in the fig. 9

All base components of device (including processor and different memory blocks) are presented in fig. 10.
Since our model has all memory hierarchy levels of the device, developers can identify which elements from application model can be placed in one of these levels. The next subsection depicts the variable allocations which are the aim of this placement. The aim is to construct an structured model that allows allocating data in the memory according to the application needs.

6.3. Variable Allocations

A critical problem in application modeling based on MDE is to manage the memory allocation in the target platform. MARTE profile adds the *flowPort* stereotype to UML *port* element. The main attribute that is aggregated on port element is the *direction*, which allows to define if the port is input, output, or bidirectional. This information contributes for deciding which elements are read-only. The other information to define a variable are provided from elements described in *MemoryMapping* metamodel (section V).
By using UML links we can associate *flowPorts* to memories in architecture models. Each port has attributes and associations that permit us to define size and data type for example. Thus, developers can define their application models where the data will be stored and how much size the data will take. In the fig. 11 we can see a simple example of allocation. The ports of the *k* instance (from Kernel application component) are allocated in *local* instance (from LocalMemory architecture component). Then, using the *memory mapping* transformation (that is part of the chain of transformations applied to the input model), we create a structured element tree having all information about size, type and, mainly, where data will be placed in a spatial or temporal way.

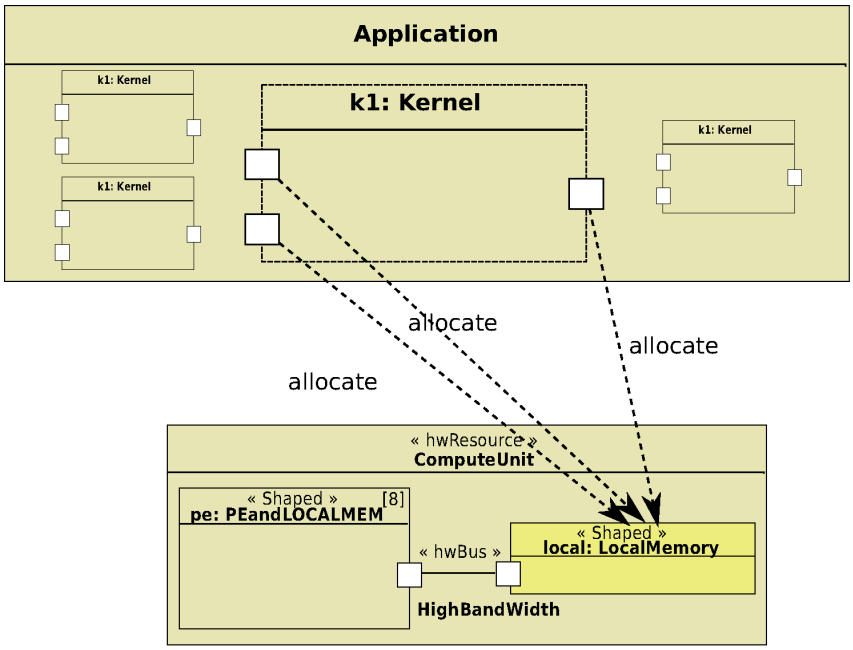

FIGURE 11 – Flowport Assotiation

7. Environment and Case Study

The architecture metamodel for GPUs is used within the transformation chain in order to create an hybrid model which is closer to the target code. EMF is the Eclipse Modeling Framework used by IBM's open source Eclipse project [7]. We used Eclipse platform to specify the memory mapping metamodel and models for GPU architecture used in this work. The DaRT team has made a new MARTE metamodel based on MARTE profile for UML. This EMF metamodel provides classes and other elements that allow us to create models which describe the applications and hardware characteristics, and the relation between them. The main goal is to define the maximum number of information about the application (and its parallelism) and hardware, thus we can create transformation rules for generating models and source code. This environment provides some tools like QVTO and Acceleo. The first one is the language from the Meta Object Facility Query/View/Transformation (MOF QVT) [13] specification. QVT is the solution for model transformations in the OMG modeling framework. This solution is a defacto standard for model transformations. QVT Operational (QVTO) is a completely imperative language, which only supports model transformation scenarios in an unidirectional M-to-N fashion. The second one (Acceleo)[12] is a non-standard solution for model-to-text(or code) transformation. Both are used in the transformation chain implemented on *Gaspard2* environment, whose the main objective is code generation for embedded systems.

The approach presented in this paper was inserted in the Marte to OpenCL transformation chain [11]. This chain allows us to create an application and test a generated code. The application in the test environment is the conjugate gradient algorithm implementation. The conjugate gradient (CG) method[9] is often used in modeling and simulation of electrical systems. It should only be applied to systems that are symmetric or Hermitian positive definite. Input data are resulting from a FEM model of an electrical machine. The matrix is stored in *Compressed Sparse Row (CSR)* format having N=132651 and NNZ=3442951. The CG algorithm is modeled in MARTE as presented in the figure 12, where data reading and initial configurations are defined by stereotyped blocks. Highlighted gray blocks represent tasks, which are mapped onto as many devices as we want to distribute the task job. Tasks, such as DGEMV(sparse), are repetitive and, thus, potentially parallel. The CGLoop is a 132651 loop which some of its input data are recovered between continuous iterations. A *continue-condition* is specified by a constraint attached to

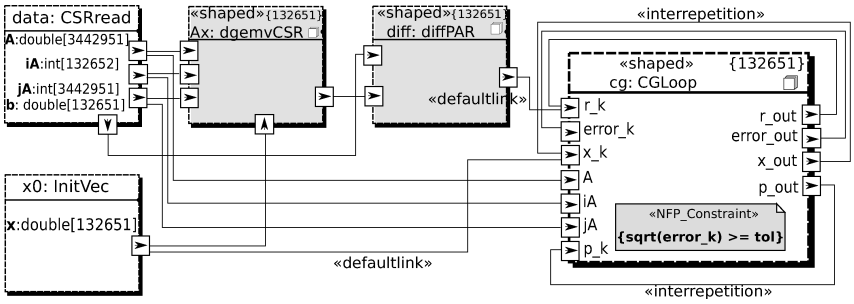

FIGURE 12 – UML/MARTE Model for Setup and CG Overview

the CG block, so the loop can stop before running all iterations. The figure 13 is an internal view of the CGLoop modeled in the figure 12. Here scalar operations run on CPU processor, and repetitive operations run on GPU processors. Details about deployment of elementary tasks (operations), data and task allocation to architecture, scheduling, grid definition, and so on, can be found in [8] and they are not discussed in this paper due to scope and space limitation.

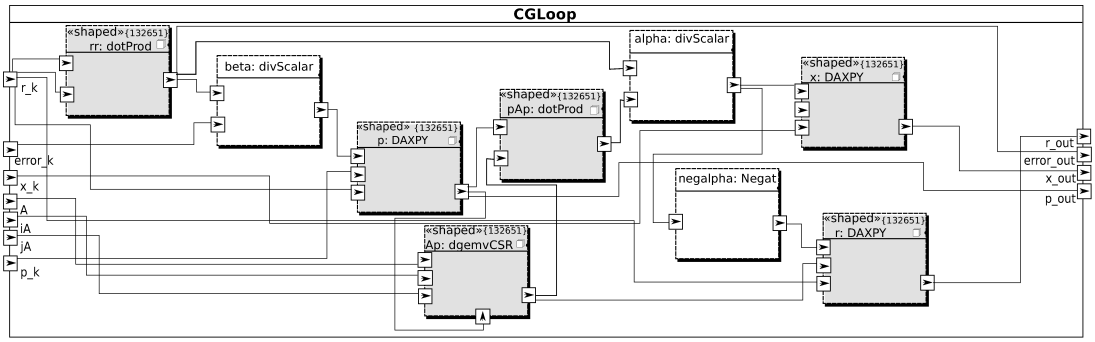

FIGURE 13 – Conjugate Gradient UML/MARTE Model

We used four double-precision implementation versions of CG. The first one (the reference result) is sequential and uses the Matlab's *pcg* function. The other ones are automatically generated OpenCL implementations whose kernels are launched onto 1, 2 and 4 devices, respectively. The number of used devices depends of the task allocation process. The hardware used is composed by a *2.26GHz Intel Core 2 Duo* processor and S1070 unit (4 Tesla T10 Nvidia GPU). Usually, manually written codes have better performance than automatic ones. However, these automatically generated CG implementations have an expressive performance(table 1) compared to sequential code (time results include just computing and data transfer times in CG loop). The multi-GPU aspect is verified in the two latest versions. The code generation compiler decides equally the task partitioning to the multiple devices. The gain is not linear(though significant) due to extra data transfers among cpu and devices. A detailed analysis about solvers and Multi-GPU can be found in [6].

8. Conclusions

The result model based on MARTE profile and metamodel extensions allowed us to describe specifical details of GPUs. Our approach presents a solution to specify a GPU architecture in terms of processor and memory hierarchy validated by a case study. As applications for embedded systems are evolving

conjugate gradient	#iter	time(s)	speed-up	gflops
Matlab PCG	117	3.17	1	.303
OpenCL (1 GPU)	116	0.659	4.81	1.45
OpenCL (2 GPU)	116	0.461	6.87	2.07
OpenCL (4 GPU)	116	0.380	8.34	2.50

TABLE 1 – Performance Results ; N=132651, NNZ=3442951, tol=1e-10

fast, MDE approaches are more suitable to these systems. Advances in GPU architecture modeling allow developing model-to-model and model-to-code transformations. These transformations contribute to generate executable code for architectures based on GPU.
An important contribution of our approach is in the memory modeling aspect. GPUs have inherent characteristics that require a optimal data allocation from developer. These requirements imply better data alignment and bandwidth use for obtaining application performance. Future works will explore these metamodel and model to generate code more efficient.

References

1. Modeling and Analysis of Real-time and Embedded systems (MARTE). `http://www.omgmarte.org/`.
2. NVIDIA CUDA Compute Unified Device Architecture. `http://www.nvidia.com/cuda/`.
3. OMG's MetaObject Facility. `http://www.omg.org/mof/`.
4. OpenCL - The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.
5. Jean Bézivin and Olivier Gerbé. Towards a precise definition of the omg/mda framework. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 273–, Washington, DC, USA, 2001. IEEE Computer Society.
6. Ali Cevahir, Akira Nukada, and Satoshi Matsuoka. High performance conjugate gradient solver on multi-gpu clusters using hypergraph partitioning. *Computer Science - Research and Development*, 25 :83–91, 2010.
7. Eclipse. *Eclipse Modeling Framework*, 2011. `http://www.eclipse.org/emf/`.
8. A. Gamatié, S. Le Beux, E. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J-L. Dekeyser. A model driven design framework for massively parallel embedded systems. *ACM Transactions on Embedded Computing Systems (TECS). (to appear)*, 2011.
9. Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 3rd edition, October 1996.
10. Holger Lange, Florian Stock, Andreas Koch, and Dietmar Hildenbrand. Acceleration and energy efficiency of a geometric algebra computation using reconfigurable computers and gpus. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0 :255–258, 2009.
11. A. Wendell O. Rodrigues, Frédéric Guyomarc'H, and Jean-Luc Dekeyser. An MDE Approach for Automatic Code Generation from MARTE to OpenCL. Technical report, INRIA Lille - RR-7525. http ://hal.inria.fr/inria-00563411/PDF/RR-7525.pdf/.
12. Obeo. *Acceleo - Model to Text transformation*, 2011. `http://www.acceleo.org/`.
13. OMG. *M2M/Operational QVT Language*, 2011. `http://wiki.eclipse.org/M2M/QVTO/`.